AN OUTLINE OF TCD COMAL PACKAGES

Colleen Kitchen

ABSTRACT

'Packages' were proposed in 1981 by the Trinity College Working Group on COMAL. The concept has caught on, although the semantics of packages were never completely specified before. The purpose of this paper is to define the package, the circumstances under which packages may be loaded and freed, the communications between packages, and a framework for implementation.
Keywords: package

1. INTRODUCTION

COMAL is a derivative of structured BASIC, except that it has failed to inherit some of BASIC's flaws (e.g. 'falling into' code for subroutines). For a description of various COMAL implementations, see [1], [2], and [3]. COMAL allows both 'open' and 'closed' procedures. Closed procedures may only communicate with the main program through parameters, and through explicit sharing of global variables. Parameters may be passed by reference or by value. (Value is the default.) A closed procedure is allowed to access named global variables by means of an IMPORT statement. (Technically, IMPORT is actually IMPORT and EXPORT, as closed procedures are allowed to modify global variables as well as look at them.) Any non-parameter, non-imported variable encountered in a closed procedure is assumed to be local to the procedure. Open procedures have no local variables. They automatically import/export non-parameter variables from the global environment without having to make it explicit. When one writes an open procedure or a closed procedure with IMPORT, one knows what variables are being brought in. This is known as static binding strategy.

'Packages' were proposed by the Trinity College Working Group on COMAL as:

1. a flexible and general means of extending the language;
2. a means of providing a sensible overlay facility;
3. a means of encouraging modular program development.

The package is an attractive idea, but care must be taken with the its semantics, else atrocities will result.

2. WHAT IS A PACKAGE?

A package consists of: (a) a uniquely labelled set of procedures, (b) an optional set of data structures, and (c) an optional initialisation procedure to be automatically 'run' whenever the package is loaded. Packages are brought into the program's workspace by issuing a 'USE' command, and are taken away by using a 'DISCARD' command. A procedure in a package may be open or closed. Procedures within packages may be 'exported', i.e. made available for use outside the package. A package procedure may call procedures from other packages, but every such call must be qualified (as explained below). The data structures and initialisation module in a package are private. Import statements in closed package procedures can only refer to data within the package. Likewise, non-local references in open package procedures can only refer to variables within the package.

For eight- and sixteen-bit machines, packages should be separately compiled and linked in under program control. The alternate or 'library' approach would be to link all necessary packages statically before executing the program. This approach, while easier, makes for a larger 'executable' module. If overlaying is to be catered for, some sort of overlay descriptor language is demanded. The dynamic approach makes the implementation somewhat trickier, but has the advantage of only including modules when the programmer thinks he needs them, (implying a smaller image), and making the overlaying of modules transparent. On a virtual memory machine, the 'library' approach could be taken with no loss of compatibility.

A COMAL program may be converted into a package (provided it meets the criteria) by means of a package builder. The package builder itself is a package, as it is relatively independent of the main COMAL system.

3. SYNTAX

The syntax of the package commands is:

⟨use statement⟩ ::=
    USE ⟨identifier list⟩
⟨discard statement⟩ ::=
    DISCARD ⟨identifier list⟩
⟨identifier list⟩ ::=
    ⟨identifier⟩ {,⟨identifier list⟩}

The package commands can be issued either from a program, or at 'system' level. Each identifier, in addition to being syntactically valid, must correspond to a physical package. Calls to package procedures look the same (except as explained below) as ordinary calls to user written routines.

4. RECURSION IN USE AND DISCARD

What if a routine inside package A contains a statement 'USE A'? Likewise what if a package tries to discard itself? Clearly, neither of these situations is desirable. There is no point in loading multiple copies of the same package, and if a package discards itself then the program turns into rubbish. The package-building module 'knows' the name of the package; hence, it can trap direct recursion.

What about indirect recursion? That is, suppose a package A contains a routine with a call to a routine in package B. The routine in package B tries to discard package A. This cannot be detected by the package builder, as it cannot 'know' what packages are going to call each other in which order. When package A's space is freed, the package B routine returns to garbage. This cannot be allowed. Can discarding be allowed at all? Yes, if the master package name table is provided with a reference count which is incremented each time a routine from the package is called, and decremented each time a routine from the package is exited.

Otherwise, attempts to 'USE' a previously loaded package, or 'DISCARD' one which is not loaded are harmless errors, as the package loader can easily detect this situation and ignore these commands.

5. QUALIFIED REFERENCES

What if Package A and Package B both contain a routine called X? If both are loaded and the interpreter encounters a call to X, which one should it be? Clearly the answer to this is to extend the syntax to allow procedure names to be qualified by package names. A qualified reference is unambiguous, since packages must have unique names. References to non-local procedures from within packages must be qualified, because the package writer does not know what other packages may be loaded by the user program which calls his package. He is therefore obliged to make such references unambiguous. Naive users writing main programs do not need to know about qualifiers, since they know what packages they are loading and what routines are available.

When the line-by-line Compiler (LBLC) encounters a qualified reference to a function, it generates a special 'qualified function' code. It also makes an entry into the package name table, thereby assigning the package its package number prematurely. Once it has the package number, then this is put into the code, along with the variable. (This is also necessary in order to be able to list the program before running it.) When the interpreter encounters one of these codes, it:

1. goes immediately to the names table in the current package and gets the name;
2. goes to the names table of the indexed package and attempts to match the name. If the match fails, then a run time error results.

We have adopted the following syntax extension for qualifiers:

⟨procedure identifier⟩ ::=
[⟨identifier⟩ :] ⟨identifier⟩

6. HANDLING OF PROCEDURE NAMES

When the LBLC encounters an unqualified procedure/function name (henceforward just called a function name – for the purpose, of this discussion functions and procedures are equivalent), it cannot 'know' whether or not the function is to be defined in a package, so it creates a symbol table entry for the function, and sets it to undefined.

When the interpreter encounters an unqualifed procedure reference, it will first locate the index in the main symbol table. If the corresponding variable is flagged as undefined, then the interpreter retrieves the procedure name from the main names table. It then successively accesses the names table for each package until it either runs out of packages (an error), or encounters a match. Once a name match is found, then the corresponding local package table is accessed for the location of the code. The first match found is taken; hence, the behaviour is undefined when multiple routines with the exact same name are loaded.

7. SCOPE

Package data cannot be accessed directly by name, but they may be read/written via routines in the package, which 'know' about them when they are compiled. This allows the representations of abstract data types to be 'hidden' from the package user.

In TCD COMAL, routines are automatically hidden by putting them into packages. In order for package routines to be accessible from outside, they must be explicitly exported.

8. LISTING OF PACKAGES AND THEIR LOCAL SYMBOLS

The listing of packages as a matter of course is not recommended; however, (from experience) we realise that one cannot forbid it entirely. Whether or not to allow a package to be listed is a parameter set by the package designer. When it can be listed, a disk access will have to be made in order to bring in the package's local symbol table, which otherwise would not be loaded. The fact that the user is able to view the package's internal symbols does not imply that he can manipulate them.

9. TYPES OF PACKAGES

For speed reasons, machine code packages, in addition to packages written in COMAL, are desirable. Package building and accessing are designed to support them. (Any package written in a compiled HLL other than COMAL is considered

to be a machine code package, since it ultimately reduces to machine code.) There are essentially two types of machine code package: relocatable and (especially for small machines) absolute. The package master table indicates for each package whether it is COMAL, relocatable or absolute machine code.

10. STORAGE MANAGEMENT

A machine code package has exactly the same format in memory as a COMAL one, except for the actual code. A COMAL package contains only offset addressing, and is 'relocated' dynamically via a pointer in the master package table. Hence, it can reside anywhere. This is not the case with machine code if we actually want to run it on the hardware. Certain memory pages (usually in low memory) need to be kept free for absolute packages; this is an area which must be tailored individually for each machine. The package loader must be individually tailored for each machine in order to make use of the relocation information provided by the assembler (or other available system software). Each new package is loaded physically in higher memory addresses than previously loaded packages, but physically below the main code, since the main code might dynamically grow. When a package is discarded its space is freed, leading to fragmentation. COMAL packages contain only relative addresses, and may therefore safely be shifted. Absolute packages obviously cannot be shifted. Relocatable machine code packages, once loaded, cannot safely be shifted, because they might contain absolute pointers as data. If memory becomes fragmented to the point where not enough contiguous space can be found to load the next package, then the program cannot continue.

11. BUILDING MACHINE CODE PACKAGES

The package-builder module for machine code packages is system dependent. The dependency rests on the type of output produced by the system assembler [compiler]. In the case where the system can produce a symbol table, the package builder reads through the assembler/compiler output file, using selector functions to access the relevant data. These selectors are individually tailored for different hardware, so that the main program does not have to be modified. It stores the code, relocating if necessary, and ignores any other information that might be in the file. When it gets to the symbol table, using selectors, it notes down all the symbols, their locations, whether or not they are global, and whether or not they are defined. It then internally reformats this information to suit its own symbol table specifications.

12. SUMMARY

Packages are defined to be essentially closed modules which are self-contained, and communicate with the outside world only through parameters. They may be used and discarded freely, as the COMAL system detects multiple 'USE's null 'DISCARD's, and the situation when the package to be discarded is still active. They communicate internally in the same ways that an ordinary COMAL program communicates with itself, but their internal workings are private. Packages grow naturally out of ordinary programs which meet certain communications criteria. They are built with the help of an interactive package-building module. Packages encourage modularisation, allow for easy extension of the language, and make overlays transparent. Source COMAL may be easily ported to different hardwares, while full advantage is taken of each individual hardware via machine code packages. The design enables the user to run a package without knowing whether it is in machine code or COMAL, owing to the fact that only entry points are exported.

13. ACKNOWLEDGEMENT

Thanks to Brendan Lynch, David O'Neill, John Byrne, and the other members of the Comal group for their respective contributions to this paper.

REFERENCES

[1] Borge Christensen, *Beginning COMAL*, Ellis Horwood, Chichester, 1982.

[2] Len Lindsay, *The CBM COMAL Handbook*, Reston Publishing (USA), to appear August 1982.

[3] Roy Atherton, *Structured Programming with COMAL*, Ellis Horwood, Chichester, 1982.

Colleen Kitchen is a Lecturer in Computing at Trinity College, Dublin.

MAGIC SQUARES

Continued from page 7.

```
PROC showsolution
   EXEC fillcells
   EXEC printboard
   FOR numberofmoves:=1 TO par-2 DO
      EXEC hitkeytogo
      MOVE:=solutionmove(numberofmoves)
      EXEC altercells
   NEXT numberofmoves
   EXEC hitkeytogo
ENDPROC showsolution
```

Brian Grainger is a mathematician with British Aerospace Ltd.